A Massively Parallel Multireference Configuration Interaction Program: The Parallel COLUMBUS Program

HOLGER DACHSEL* and HANS LISCHKA†
Institut für Theoretische Chemie und Strahlenchemie, Universität Wien, A-1090 Vienna, Austria

RON SHEPARD
Theoretical Chemical Group, Chemistry Division, Argonne National Laboratory, Argonne, Illinois 60439

JAROSLAW NIEPLOCHA and ROBERT J. HARRISON
Pacific Northwest National Laboratory, Richland, Washington 99352

Received 19 February 1996; accepted 26 April 1996

ABSTRACT

A massively parallel version of the configuration interaction (CI) section of the COLUMBUS multireference singles and doubles CI (MRCISD) program system is described. In an extension of our previous parallelization work, which was based on message passing, the global array (GA) toolkit has now been used. For each process, these tools permit asynchronous and efficient access to logical blocks of 1- and 2-dimensional (2-D) arrays physically distributed over the memory of all processors. The GAs are available on most of the major parallel computer systems enabling very convenient portability of our parallel program code. To demonstrate the features of the parallel COLUMBUS CI code, benchmark calculations on selected MRCI and SRCI test cases are reported for the CRAY T3D, Intel Paragon, and IBM SP2. Excellent scaling with the number of processors up to 256 processors (CRAY T3D) was observed. The CI section of a 19 million configuration MRCISD calculation was carried out within 20 min wall clock time on 256 processors of a CRAY T3D. Computations with 38 million configurations were performed recently; calculations up to about 100 million configurations seem possible within the near future.

*Present address: Pacific Northwest National Laboratory, Richland, WA 99352.
†Author to whom all correspondence should be addressed.
E-mail: lischka@itc.univie.ac.at

Journal of Computational Chemistry, Vol. 18, No. 3, 430–448 (1997)

CCC 0192-8651 / 97 / 030430-19

Introduction

Parallel computing is one of the great challenges in quantum chemistry. It has a relatively long tradition in the computer sciences but was originally rather slowly accepted in computational chemistry. The reasons for that are manifold, but one main obstacle to a broad acceptance was certainly the fact that in the early days of parallel computing the hardware equipment was usually very limited (e.g., in terms of central memory and bandwidth of the interprocess communication). Operating systems and software tools were also still in their beginning. Because quantum chemical programs are very demanding in all aspects of computer resources, only very idealistic investigators could be convinced to spend work on parallelizing their programs. The situation has changed significantly since then. Not only has the hardware situation improved strongly, but also the demand for quantum chemical calculations has increased dramatically. There is quite general agreement that the required computer power can only be obtained by parallel computing. However, there is much less agreement on the strategies of how to actually proceed. Even though the software situation has improved significantly, standards (like MPI[1]) are only becoming generally implemented now. Emerging language standards such as high performance Fortran[2] are still far from being widely accepted. The fact that many computer companies have developed their own parallel computing tools is also not very helpful in the long run considering portability problems and the rapid fluctuations encountered with those tools.

Starting with the pioneering work by Clementi and coworkers[3] and others[4–6] on a ''loosely coupled array of processors (LCAP),'' many investigations originally concentrated on the parallelization of self-consistent field (SCF) programs[7–15] because of the importance, but also because of the simplicity of the method. Under the condition of storing local copies of the Fock and density matrices on each processor (replicated data approach), the parallelization of the Fock matrix construction step within the direct SCF scheme can be done effectively. Recently, very promising attempts were also made to go beyond the limitations of the replicated data method.[16–18] In the last few years very successful efforts to parallelize electron correlation methods such as Møller–Plesset perturbation theory,[19–21] coupled cluster,[22,23] two-electron integral transformations, Multi Configuration SCF (MCSCF), and configuration interaction (CI) methods[24–29] were undertaken. For an overview about recent investigations and developments see the extended reviews by Harrison and Shepard[30] and Kendall et al.[31]

The purpose of the present investigation is the continuation of our previous work[32–34] on the parallelization of the general multireference singles and doubles CI (MRCISD) method based on the COLUMBUS program system.[35–37] General MRCI programs are usually very complex and consist of many major computational steps like SCF, MCSCF, two-electron integral transformation, and determination of the lowest eigenvalues and corresponding eigenvectors (''diagonalization'') of the Hamiltonian matrix. Each of the just-mentioned program sections can be split further into still quite extended and complicated subsections. Thus, the parallelization of such a large program system is a major undertaking and can only be carried out in steps. At present we have finished the parallelization of the diagonalization section of the COLUMBUS program, which is the most complicated, and in extended MR cases, also by far the most time consuming part of the whole program.

Two major ideas formed the basis of our work: we wanted to develop flexible algorithms that should lead to a massively parallel MRCI program and the programs should be easily portable.

Review of Our Previous Work

In this section an outline of the basic features of the sequential program and a review of our previous work[32–34] on the parallelization of the COLUMBUS program system[35–37] are given. The COLUMBUS program is a collection of Fortran programs for performing general *ab initio* MRCISD calculations. It consists of an atomic orbital (AO) integral, SCF, MCSCF, and CI program sections. In addition to the CI method, various variationally oriented approaches, including size-consistency corrections such as average coupled pair functional (ACPF)[38] and MR-average quadratic coupled cluster (MR-AQCC),[39] are included. As already mentioned above, we devoted our parallelization efforts so far to the CI diagonalization part of the program system. Therefore, this review is restricted only to those aspects directly related to this topic. COLUMBUS is based on the graphical unitary group approach (GUGA).[40,41] Within

GUGA each configuration state function (CSF) is represented as a walk on the Shavitt graph, the latter being a graphical representation of the distinct row table (DRT). The Davidson diagonalization method[42] is used to determine the lowest eigenvalues and corresponding eigenvectors of the Hamiltonian matrix **H**. In this scheme, the computationally most important step is the calculation of the sparse matrix–vector product $\mathbf{w}_i = \mathbf{H}\mathbf{v}_i$. The $\mathbf{v}_i$'s are the expansion vectors for the CI wave function and the $\mathbf{w}_i$'s are the resulting product vectors. A ''direct CI'' procedure[43,44] is used to compute this matrix–vector product. It is driven by the four indices of the two-electron integrals. The contributions of the integrals to $\mathbf{H}\mathbf{v}_i$ are called CI coupling coefficients or GUGA loop values. Similar to the work of Siegbahn,[45] these coefficients are factorized into internal and external orbital contributions where orbitals are classified as internal ones if they are occupied in at least one of the reference configurations and as external ones otherwise. In the sequential program and in the first parallel implementation the internal parts of the coupling coefficients are computed beforehand and stored on a file (''formula tape''). The external contributions are computed on the fly as needed.[46]

The quadruple of the two-electron integral indices is classified according to the number of external indices. Each of the five types (0–4 external) is treated separately. The integrals and the coupling coefficients are arranged such that the contributions to $\mathbf{H}\mathbf{v}_i$ can be calculated by dense matrix operations of matrices or vectors of the dimension of the molecular orbital (MO) basis. In the original sequential and also in the parallel program the expansion and update vectors are divided into segments. The segmentation follows the classification of the CSFs according to all-internal configurations (Z walks), singly excited (Y walks), and double excited (X walks and W walks) configurations.[41] In the latter case, the two electrons occupying external orbitals are triplet coupled for the X walks and singlet coupled for the W walks. The Z and Y walks are combined into one segment (this restriction has been partially removed, see later in the text) and always kept in central memory. It is only the X and W walks that were divided into segments. Each segment contains walks of a given type (W or X) only. The segmentation was carried out at the level of internal walks in the DRT. Thus, all external continuations of an internal walk (i.e., all excitations belonging to the same internal configuration) are stored in the same segment. Utilizing the symmetry of **H** in computing the matrix–vector product **Hv**, a loop over all segment pairs is performed (see also fig. 2 in ref. 32):

```
do seg i = 1,nseg
  read v_seg i, w_seg i
  do seg j = 1,seg i
    read v_seg j, w_seg j
    update w_seg j, w_seg j
    write w_seg j
  enddo seg j
  write w_seg i
enddo seg i
```

The procedure ''update'' stands for the updating process of **Hv** into **w** for a segment pair (seg i,seg j). The segment structure is the same for all trial and update vectors. The **v** and **w** vectors were originally stored on disk and only the required segments were transferred into central memory.

In the subspace section of the program the matrix elements $\tilde{H}_{ij} = \mathbf{v}_j^t \mathbf{H}\mathbf{v}_i$ of the Hamiltonian matrix in terms of the subspace expansion vectors $\mathbf{v}_i$ are calculated from $\mathbf{H}\mathbf{v}_i$, and a new expansion vector is generated.

A coarse-grain parallelization of the $\mathbf{H}\mathbf{v}_i$ step was performed (for details see ref. 32) in which the outermost loops over the segment pairs shown above were used for parallelization. This choice had several advantages:

- it provided the most coarse-gain decomposition possible;
- the considerable complexity of the code below these loops did not affect the parallelization;
- the number of tasks scaled with the square of the number of segments. Thus it was possible to generate a sufficiently large number of tasks to make load balancing effective.

Dynamic load balancing was implemented via a shared counter. Each index in this counter corresponded to a task defined as the work to compute the contribution of one segment pair to the matrix–vector product $\mathbf{w} = \mathbf{H}\mathbf{v}$. In the spirit of a replicated data approach, each process had a local copy of **v** and **w** and had to read all integrals (and other quantities, including indexing arrays, which were not important in terms of input/output, I/O). The reading of the formula tape from a file (as done in the sequential program) was replaced by a straightforward recomputation of the loop values. After completion of the loops over segment pairs

the total vector **w** was obtained by a global sum operation from all partial contributions that had been accumulated by each process. The subspace section had not been parallelized at all.

The just described parallel algorithm[32] was implemented by using message passing based on the portable program package TCGMSG[47] developed by one of us (R.J.H.). This program version was ported to a variety of parallel computers, both shared memory (Alliant FX/2800, CRAY Y-MP, Convex C2) and distributed memory (iPSC/860).

The main purpose of our first implementation was to investigate the efficiency of our parallelization strategy. The load balancing scheme worked very well and substantial speedups could be achieved for calculations using up to eight processors. The most serious drawback was the strict data management scheme that was imposed by the synchronous message passing framework. As already mentioned, all data had to be distributed to all processors at the beginning of the calculation (two-electron integrals) or at the beginning of each Davidson iteration (trial vector), respectively. Because of the dynamic load balancing scheme it was not clear, however, which segment pairs would be assigned to which processors. Therefore, all data had to be broadcast to all processors even though only a fraction of it was actually used. As a consequence, the time for broadcasting the data at the beginning and for the global sum at the end of each **Hv** step started to become a serious bottleneck (see table 1 in ref. 34). Moreover, due to the replicated data approach the calculations were severely limited by central memory requirements.

Outline of New Parallel CI Program

From an analysis of the situation as discussed just above it was clear to us that a major improvement could only be achieved if asynchronous one-sided access to data distributed over the central memory of all processors were possible in the sense of a ''virtual shared memory.'' In this way two bottlenecks could be removed at once: broadcasting of all data at the beginning of each Davidson iteration and collecting the results via a global sum could be avoided because all necessary data were available just in time as needed, and the restrictions of the replicated data approach could be removed by storing just a single copy of each array.

For that purpose we used the global array (GA) toolkit developed recently by Nieplocha and colleagues.[48] These tools give us exactly those possibilities we need in terms of functionality and portability. The key concept of the GA is that it provides a portable interface through which each process in a multiple-instruction multiple-data (MIMD) parallel program can independently, asynchronously, and efficiently access logical blocks of physically distributed matrices, with no need for explicit cooperation by other processes. Each section of a distributed matrix is logically assigned to exactly one process that is assumed to be able to access data in this section faster than data in the remainder of the matrix. The GA toolkit implements a shared-memory programming model in which data locality is managed explicitly by the programmer. The management is accomplished by explicit calls to noncollective functions transferring data between a global address space (a distributed array) and local storage. In this respect GA is similar to distributed shared memory (DSM) models. However, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be explicitly specified and used. Global arrays were designed to complement rather than replace the message-passing programming model. The programmer is free to use both the shared memory and message passing in the same program and to take advantage of existing message-passing software libraries. A typical application would have the following structure: after a memory allocation and initialization step, 1- and 2-dimensional (1-D, 2-D) arrays can be distributed over the memory of all processors by calling first the routine *ga*_create that breaks the whole array in *n*proc blocks (*n*proc is the number of processors) and assigns these blocks to the individual processors. The routines *ga*_put, *ga*_acc, and *ga*_get are used for storing, accumulating, and retrieving data. Besides these basic features there are more GA operations available like scattering elements into an array and gathering elements from an array or data-parallel basic linear algebra subroutine (BLAS)-like routines. For more details we refer to ref. 48.

In addition to GAs we still have the option to store multiple copies of certain data files (like the two-external integrals) in central memory because these files are relatively small but frequently accessed. This local data storage device is called virtual disk. A special data management scheme was developed for that purpose. Storage space is simply allocated in the form of arrays by standard

Fortran syntax. This space is unique to the application program running on a particular processor.

As compared to our first parallel program version, a number of features had to be added or improved to make the use of larger processor numbers more efficient. The most important developments concern a generalized task list, and the parallelization of the GUGA loop construction and of the Davidson subspace operations.

Task List

In the original parallel program a task was defined as the entire work to be done for a given segment pair. To achieve more flexibility in defining tasks this work is split further into subcases according to the number of internal indices (0–4 internal) of the two-electron integrals. Furthermore, the segment lengths can now be chosen differently for the 0- and 2-external, 1-external, and 3- and 4-external integral cases. Previously, the Y and Z walks were treated as one segment and kept as local copies on each processor. For larger active orbital spaces the Y and Z walks were also segmented for some task types. For the purpose of documentation the segmentation scheme defining the tasks is shown in Table I. This scheme is based on an analysis of the GUGA loop structure in a MRCISD calculation as given by Shavitt.[46] In this table we list the integral type and the type of the pair interaction according to which the task is characterized. The number of tasks for each type is given by the number of segment pairs listed in Table I. The number of segments *n*seg *XW*34 applies to the treatment of the 3- and 4-external two-electron integral cases, *n*seg *XW*02 to the 0- and 2-external two-electron integral cases, and *n*seg *XW*1 to the 1-external two-electron integral cases. The segmentation is carried out such that the lengths of all segments for a given type (e.g., X or W walks) are approximately the same. The actual choice for the number of segments is related to the number of processors and will be discussed below.

Based on this segmentation scheme a task list is generated with one entry for each segment pair interaction. The number of entries is the sum of all pair interactions given in the last column of Table I. The tasks can be worked on by the program in any order. Usually, we use an ordering according to execution times (see later).

Data Transfer Considerations

A classification of data access dependent on the segmentation scheme is useful for an assessment of the resulting interprocess communication within the network. From Table I one can see that the

TABLE I.
Characterization of Tasks.

Type No.	Integral Type	Type Segm. Pair Interaction	No. Segm. Pairs
1	4-ext.	X/X[a] or W/W[a]	*n*seg*XW*34
2	4-ext.	Y/Y[b,c]	1
3	3-ext.	Y[b]$/X$ or Y[b]$/W$	*n*seg*XW*34
4	2-ext.	Y/Y[d]	*n*segY(*n*seg*Y* + 1) / 2
	and 0-ext.	Y/Y[d]	
5	2-ext.	X/X or W/W or X/W	
		X/Z[e,f] or W/Z[e,f]	*n*segXW02(*n*seg*XW*02 + 1) / 2
	and 0-ext.	X/X or W/W	
6	1-ext.	Y[b]$/X$, Y[b]$/W$	*n*seg*XW*1
7	1-ext.	Y[d]$/Z$[f]	*n*seg*Y*
8	0-ext.	Z/Z[g]	*n*seg*Z*(*n*seg*Z* + 1) / 2

[a] Walks differ only in external orbitals. Thus, only interactions within one segment occur.
[b] *Y* walks as one segment, stored in central memory.
[c] 4-External one-electron integrals only.
[d] *Y* walks segmented, stored in central memory.
[e] Contribution included in the *X*/*X*, *W*/*W*, or *X*/*W* pairs, respectively.
[f] *Z* walks as one segment, stored in central memory.
[g] *Z* walks segmented, stored in central memory.

exact dependence of data transfer will be rather complicated because of the flexible segmentation scheme. For the sake of simplicity we want to discuss here the basic behavior only and will give concrete numbers in the benchmark examples. The computation of the 4-external integral contributions can be carried out by a single sweep through the **v** and **w** segments because these contributions refer to walks that have the internal part in common. However, the complete 4-external integrals have to be read during the treatment of each segment. Because of the fact that the Y and Z walks are kept as one segment permanently in the core, the 3- and 1-external integral cases can also be treated by a single sweep through the **v** and **w** segments. A loop over segment pairs is only necessary for the 2- and 0-external integral cases. Here, all 2- and 0-external integrals have to be read while processing each segment pair. The frequency of accessing the integral files can be taken directly from Table I. Reading of the 3- and 4-external integrals, which constitute the largest fraction of the two-electron integrals, goes linear with $n\mathrm{seg}\,XW\,34$. An analogous dependence on $n\mathrm{seg}\,XW1$ is found for the 1-external integrals. From the cases depending quadratically on the number of segments, the 2-external contributions to the X/X, W/W, and X/W interactions are most important. Even though the size of the 2-external integral file is only (disregarding spatial symmetry) $3(n_{\mathrm{int}}(n_{\mathrm{int}}+1)/2)(N_{\mathrm{ext}}(N_{\mathrm{ext}}+1)/2)$ integrals, where n_{int} is the number of internal and N_{ext} is the number of external orbitals, the total amount of data transferred could become significant for larger segmentation numbers because this file is read $n\mathrm{seg}\,XW02(n\mathrm{seg}\,XW02+1)/2$ times. In many cases storage of the 2-external integrals (and, because of the modest storage requirements, also the 1- and 0-external integrals) as replicated data on the local virtual disks will be possible. In this way any data access problems concerning these integral types are largely removed. The 3- and 4-external integrals are stored as GAs unless indicated otherwise. In cases where sufficiently fast disk access is available, the 3- and 4-external integrals could also be stored there. The most important contribution to the access of the **v** and **w** segments comes from the 2- and 0-external cases (task type no. 5). Assuming exactly equal segment lengths L, the total amount of **v** and **w** elements transferred is $2L\cdot n\mathrm{seg}\,XW02^2$, which gives $2N_{XW}\cdot n\mathrm{seg}\,XW02$ where N_{XW} is the dimension of the X and W part of the CI expansion. The fact that the segments are not exactly of equal length does not change the linear dependence of reading the expansion vectors and updating the product vectors with regard to $n\mathrm{seg}\,XW02$.

In addition to the just-mentioned files, other data like indexing arrays and the DRTs have to be read for each segment pair. However, these are minor quantities and can be omitted from the discussions concerning data transfer.

The **v** and **w** vectors will be stored as GAs during the calculation. The decision where the integral files (0–4 external) will be stored can be made by setting input switches for each file separately. If the option is set to GAs, integral files will be read from the disk by one processor during startup and copied to the GAs. In case the integral file should be stored on the virtual disk, one process reads the data from disk and broadcasts them to the other processes that in turn, copy the data to their local virtual disk. The final CI vector can be copied back to the disk at the end of the calculation to save it for further purposes. It should be noted that we use the GAs only to store data that were originally sequential or direct access files organized in larger records (typically 32,768 bytes). Random access to individual array elements stored in GAs is avoided.

It should be mentioned at this point that avoiding data access collisions also turned out to be a significant factor for achieving good timings. In the original program the 3- and 4-external integrals were read sequentially starting at the beginning of the file. As a consequence all processors, to which tasks dealing with 3- and 4-external integrals had been assigned, initiated processing of integral data by requesting the first record in the integral file. This created a bottleneck because this record had been located in the memory of only one process that could only service one request at a time. To improve on the situation, each processor now starts at a different record of the integral file. These starting numbers are chosen in such a way to reduce the access conflicts at least partially.

Dynamic Load Balancing

Our dynamic load balancing algorithm for the **Hv** step is based on the task list described above. The tasks are numbered sequentially. These task list numbers create a pool of available indices. Load balancing is achieved in the following way: with the help of the TCGMSG function *n*xtval( ) (which implements a global counter variable,

available to all processes in the shared-memory style) each process is assigned the next free index from a shared counter. This index defines the task through the task list number and the necessary calculations are performed. As soon as the task is finished the processor is ready for the next available one by requesting the next free index (see fig. 3 of ref. 32). All processes work on an equal basis and no master/slave concept is used.

Good load balancing means that the granularity of the individual tasks has to be fine enough to ensure that none of the processors wait idling for a substantial amount of time at the end of the **Hv** step while others are still busy. The granularity is determined by the number of **v** and **w** vector segments. Increasing the number of segments produces a finer granularity that improves the efficiency of load balancing. The CPU overhead from increasing the segment numbers is modest. However, the data access requirements will increase with increasing segment numbers. Therefore, one has to seek a compromise between load balancing and data access aspects.

We characterize the average workload of a processor by the number of tasks $n$tppr per processor according to

$$n\text{tasks} = n\text{tppr} \cdot n\text{proc}, \tag{1}$$

where $n$tasks is the total number of tasks available and $n$proc is the number of processors. Because the computational requirements differ considerably between individual program sections dealing with the different integral types, we split eq. (1) into three parts. For the 3- and 4-external integral tasks we set

$$n\text{seg}\,XW34 = n\text{tppr34} \cdot n\text{proc}. \tag{2}$$

For the 0- and 2-external integral tasks

$$n\text{seg}\,XW02(n\text{seg}\,XW02 + 1)/2 = n\text{ppr02} \cdot n\text{proc}, \tag{3}$$

and for the 1-external integral tasks

$$n\text{seg}\,XWI = n\text{tppr1} \cdot n\text{proc}. \tag{4}$$

Appropriate values for the tasks per processor $n$tppr34, $n$tppr02, and $n$tppr1 have to be determined as described below. Then, the respective number of segments can be determined from eqs. (2) to (4). The number of segments $n\text{seg}Y$ and $n\text{seg}Z$ required for the remaining tasks are determined as the ratio of the number of the Y or Z walks, respectively, and a scaled size of the largest X or W segment of the segmentation for the 0- and 2-external integral cases. These latter tasks do not produce any data access problems at all. Therefore, they can be split into sufficiently small pieces and constitute an additional pool of tasks.

To further improve load balancing for a given number of segments, we create a task list that is ordered according to decreasing execution time in every iteration after finishing the **Hv** computation. Thus, the granularity of our tasks becomes finer toward the end of the **Hv** step, which is exactly what is needed to keep processors from idling. At the moment we do not have a procedure to estimate the execution time for a segment pair accurately enough. Therefore, we create a preliminary task list in a more or less arbitrary way and monitor the timings for the use in the subsequent iterations.

We define an efficiency of parallelization e for the **Hv** step as the ratio,

$$e = t_{\text{total}}/(n\text{proc} \cdot t_{\max}), \tag{5}$$

of the total (wall clock) time t_{total} (which is the sum over all times spent by the processors to compute **Hv**) and the time actually available for computation (including idle times), which is $n$proc $\cdot\, t_{\max}$ where $t_{\max}$ is the time for the processor that finishes last. The efficiency e is one for ideal load balancing and less than one in nonideal cases. It is a good indicator for situations where one (or more) processors start to block the calculation.

Parallelization of GUGA Loop Generation

In a direct CI program the coupling coefficients determining the contribution of the two-electron integrals to the matrix elements in **H** are needed for the evaluation of **Hv**. As already mentioned, this is done in the COLUMBUS program within the framework of GUGA. In our sequential CI program, the contribution of the internal orbital part to the GUGA loops and certain indexing information were calculated beforehand and stored in the formula tape. Even though the number of internal orbitals is only a small fraction of the external ones, the formula tape happened to be very large in extended MR cases. Therefore, in our first parallel program version, the formula tape was removed to save I/O time. Instead, the loop values were created directly in each iteration. This was achieved by simply moving the formula tape

code into the CI section of the program. It was sufficient to calculate only the loops for a given segment pair for each task. However, there was substantial overhead because the DRT used to create the internal walks belonging to one segment was constructed from a DRT representing all internal walks of a given class (Y + Z, X and W walks, respectively). Increasing the number of segments decreased the number of valid walks belonging to that segment. Thus, for a given segment, the percentage of valid walks within the DRT decreased, leading to a substantial computational overhead for the loop construction because too many invalid attempts were made to create loops contributing to a given segment pair. Therefore, we constructed separate DRTs for each vector segment. Each of these DRTs was tailored to fit the respective segment, and all vertices and arcs not used for that segment were taken out. In this way the aforementioned overhead was almost completely removed. It was sufficient to carry out this procedure for the 0- and 2-external integrals (see Table I) because these cases depend quadratically on the number of segments in contrast to the linear dependence of the remaining cases. As will be demonstrated below, the total CPU time is now practically constant with respect to the number of segments (up to a reasonable upper limit), which gives us sufficient flexibility to adjust the segmentation scheme to the number of processors.

Davidson Subspace Operations

In the Davidson subspace section of the program the projected Hamiltonian matrix $\tilde{H}_{ij} = \mathbf{v}_j^t \mathbf{H} \mathbf{v}_i$ is computed as the scalar products $\mathbf{v}_j^t \mathbf{w}_i$. Both **v** and **w** vectors are stored as GAs. To exploit data locality and minimize communication, each process computes a fraction of the scalar product corresponding to the section of GA data logically assigned to that process. Static load balancing is used and each processor has the same work to do. Only scalar quantities have to be transferred between processors. The other major tasks are the computation of the current residuum and of the new trial vector. All these operations can be performed in the same manner as the calculation of $\tilde{\mathbf{H}}$, i.e., operations on the **v** and **w** vectors are always carried out pairwise in the memory local to the processor. In this way the Davidson step could be parallelized very efficiently. The necessary computer time is usually insignificant as compared to the **Hv** step.

Startup Operations

Optimizing the startup required more attention than initially expected. Because of the efficient parallelization of the actual body of the program, the startup section started to become a problem in larger processor numbers. It was parallelized to a large extent. The following steps are worth mentioning:

- input (a few dozens of bytes) is read from a disk file by one processor and then broadcast to the others;
- the determination of the segmentation scheme for the **v** and **w** vectors (not yet parallelized);
- the diagonal Hamiltonian matrix elements are calculated in parallel and are stored as a GA;
- the two-electron integrals are read from disk by one processor and are stored, depending on the case, as GAs or on a local virtual disk as requested by input directives;
- a first trial vector is constructed (usually a unit vector)

Overall Structure of Parallel Program

The overal structure of the program is depicted in Figure 1. It follows the discussion given for the individual steps. After the startup preparations the program enters the Davidson iterations. For a given **v** vector the update vector **w** is computed. This is by far the most time-consuming program section. The synchronization step at the end of that block guarantees that all processors have finished before entering the Davidson subspace operations in which a new trial vector is generated. Another synchronization point has to be set before the program enters the **Hv** part again.

Benchmark Calculations

With the benchmark calculations we want to provide information on the efficiency of our program for typical CI cases and to demonstrate the portability of the code. Especially under consideration of the latter aspect, we present results for the

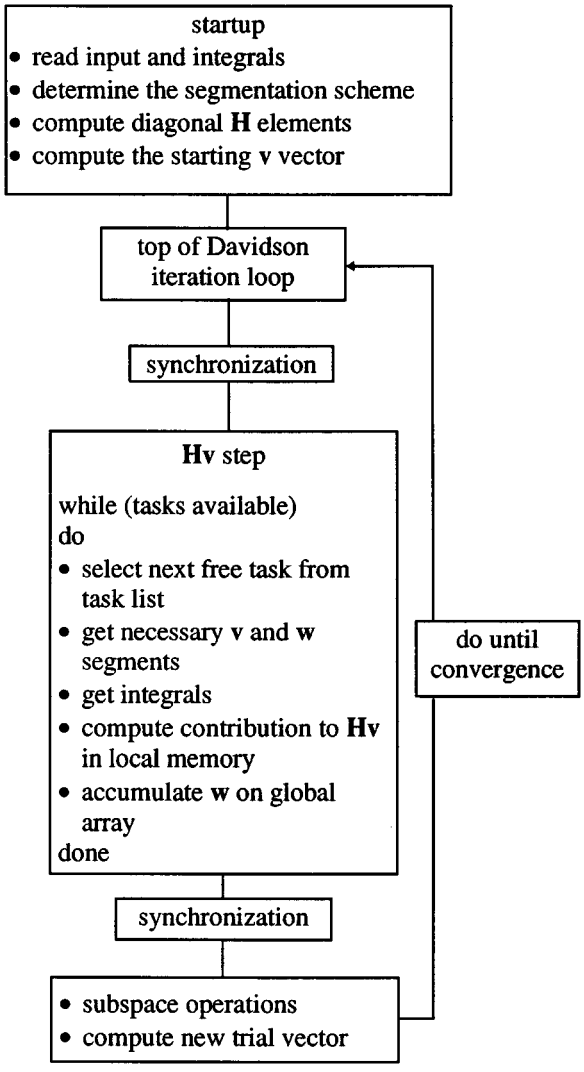

FIGURE 1. Overall structure of the parallel CI program.

CRAY T3D, the IBM SP2, and the Intel Paragon. All three computers are distributed memory machines and are good candidates to test the massively parallel aspects of the program. Concerning the evaluation of the presented timings and comparisons of different computers, see the authors' notes at the end of article.

The preferred area of usage of the present parallel CI code is extended MR cases giving large CI expansions. Two typical representatives were chosen: the methyl radical with a CAS (7/7) and the butadiene ground state with a CAS (4/5) reference space. At the other end of the application scale we selected a SRCI calculation on butadiene. For more information concerning details of these calculations see Table II.

Crucial for a good functioning of our dynamic load balancing scheme is the availability of a sufficient number of tasks to avoid idling of processors. For that purpose the numbers of tasks per processor for the three most important program sections was defined in eqs. (2)–(4). Optimal values for *n*tppr34, *n*tppr02, and *n*tppr1 depend on the type of the CI calculation and on the characteristics of the computer. The most satisfactory way of dealing with this situation would be to develop a mathematical model allowing the prediction of the execution time of the tasks. Because we need accurate estimates the model has to be quite sophisticated. This would probably mean that one had to scan at least once through the GUGA loop generation code. As an alternative, the segmentation scheme could be dynamically adjusted during the calculation. This would be possible without a major program restructuring but at the cost of efficiency because at least one iteration would run on a not so well-balanced level. Because we plan to replace the evaluation of the 3- and 4-external integral contributions by a direct scheme in the very near future, we postponed the development of the aforementioned model to the time until this new feature will be available. Until then we intend to use parameter values that are determined empirically. Of course, they will not be optimal in general. However, we are confident that with some minor observation of the task list statistics and the parallel efficiency [Eq. (5)], it will be possible to achieve timings that are close to optimal, at least for series of similarly structured calculations.

To illustrate the situation when evaluating the *n*tppr parameters, we show in Figure 2 the distribution of the number of tasks dependent on task execution times for the CH_3 CAS (7/7) case using a selection of values for *n*tppr34. The number of processors *n*proc is 64 and *n*tppr34 is set to (a) 1/8, (b) 1/4, and (c) 1/2. The values for *n*tppr1 = 1/2 and *n*tppr02 = 5 were kept constant. For all three parameter sets tasks of type 3 (3-external integrals) dominate the high end of the distribution. Moreover, for case (a), the distribution of tasks is grouped into two sets concentrated around small and large execution times. The existence of a few large tasks is sufficient to block the whole calculation. Execution of such a task takes longer than the total work available for most of the other processors. This situation is well characterized by the parallel efficiency *e*, which is only 63.4% in this case. With choice (b) the size of the largest task is significantly reduced because the number of **v** and **w** vector segments is increased. The

TABLE II.
Characterization of Benchmark Calculations.

	CH_3 CAS (7 / 7)	Butadiene CAS (4 / 5)	Butadiene SR
Basis set	cc-*p*VTZ[49]	cc-*p*VDZ[49]	cc-*p*VDZ[49]
Symmetry	D_{3h}[a]	C_{2h}	C_{2h}
No. orbitals	a_1 30, b_1 14, b_2 20, a_2 8	a_g 32, a_u 11, b_g 11, b_u 32	a_g 32, a_u 11, b_g 11, b_u 32
Reference space	CAS: 7 el. in $\{2-5a_1, 1-2b_2, 1b_1\}$	CAS: 4 el. in $\{1-2b_g, 1-3a_u\}$	Closed shell SR
Frozen core	$1a_1$	$1-2a_g$, $1-2b_u$	$1-2a_g$, $1-2b_u$
No. ref. config.	188	28	1
No. CSFs	624,334	3,933,377	82.772
	Z: 188; *Y*: 16,926	*Z*: 1,984; *Y*: 81,121	*Z*: 1; *Y*: 249
	X: 357,006; *W*: 250,214	*X*: 2,191,255; *W*: 1,659,017	*X*: 36,370; *W*: 46,152
Integr. files (MB)[b]			
4-ext., symm.[c]	4.98	6.55	7.60
4-ext., asymm.[c]	4.72	6.29	7.34
3-ext., symm.[c]	2.36	5.24	4.98
3-ext., asymm.[c]	2.36	5.24	4.46
2-ext.	0.383	1.54	1.11
1-ext.	0.0389	0.238	0.141
0-ext.	0.0020	0.0164	0.0082

[a] Only C_{2v} symmetry was used.
[b] No indices were stored.
[c] Symmetric and antisymmetric linear combinations of integral triples $(ij|kl)$, $(ik|jl)$, and $(il|jk)$.

distribution of task timings is not split into two parts anymore. A very good parallel efficiency *e* of 99.6% is obtained. In case (c) the granularity of tasks is still finer but the parallel efficiency (99.4%) is practically the same as for case (b). However, the amount of data transferred is increased further in the latter case without any benefit for the calculation. Thus, case (b) is selected as our standard choice. Along these lines, systematic investigations were made for all parameters on each computer separately. Finally, we arrived at the following recommended *n*tppr values:

CRAY T3D: *n*tppr1 = 1/2, *n*tppr02 = 5, *n*tppr34 = 1/4;
IBM SP2: *n*tppr1 = 1/2, *n*tppr02 = 2, *n*tppr34 = 1/4;
INTEL Paragon: *n*tppr1 = 1/2, *n*tppr02 = 2, *n*tppr34 = 1/4.

Using these values, CI calculations on all benchmark examples were carried out to investigate the scaling of the calculations with the number of processors. In Table III a detailed analysis of one typical **Hv** computation on the CRAY T3D is given for the CH_3 CAS (7/7) and butadiene CAS (4/5) cases. The minimal number of processors is four for CH_3 and 16 for the butadiene CAS (4/5) because of memory requirements for the GA and local virtual disks. Each node of the T3D was equipped with 64 MB of central memory. Let us first look at total times computed as the sums over the times each processor spent for calculating its contribution to **Hv**. Its dependence on increasing numbers of *n*proc measures the overhead due to the concomitant increase of the number of CI vector segments. As one can see the total time remains constant within 5% or less. One contribution to the slight increase in time comes from the GUGA loop generation already discussed above. The second contribution comes from increasing access to the two-electron integrals. To illustrate the situation further, total data transfer times, total amounts of data transferred, and effective data transfer speeds are also given in Table III. The most important part of data transfer comes from reading the **v** and updating the **w** vector (which is not included in the total **Hv** time). As expected, total data transfer times and the amount of data transferred increase approximately with the square root of *n*proc in that case. General formulas were worked out for that and the other cases but are not given here for the sake of simplicity. Timings for reading the 3- and 4-external integrals increase

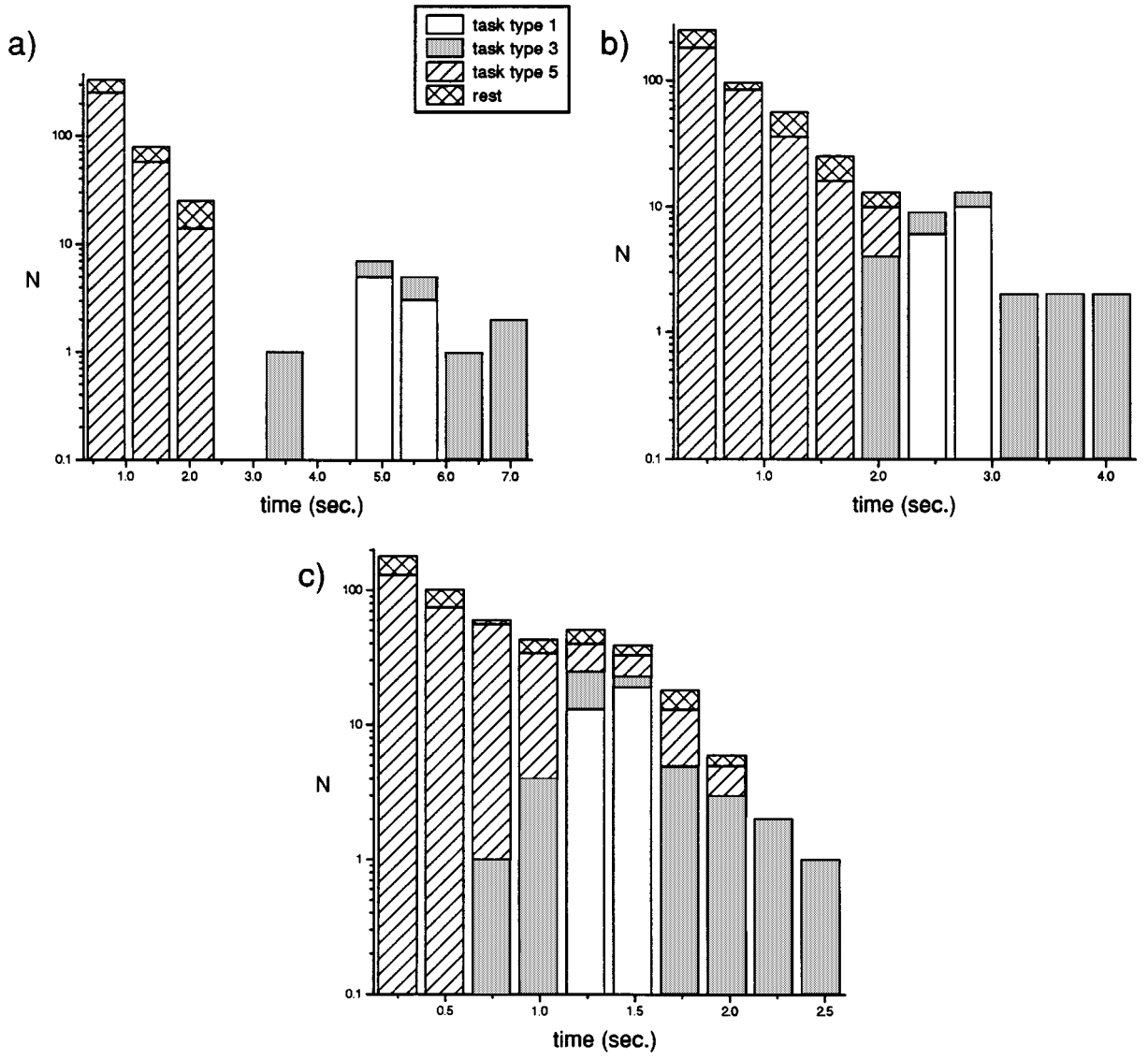

FIGURE 2. Distribution of individual task times for various sets of tasks per processor: (a) *n*tppr1 = 1 / 2, *n*tppr02 = 5, *n*tppr34 = 1 / 8; (b) *n*tppr1 = 1 / 2, *n*tppr02 = 5, *n*tppr34 = 1 / 4; (c) *n*tppr1 = 1 / 2, *n*tppr02 = 5, *n*tppr34 = 1 / 2. The timings were taken on a CRAY T3D for the CH_3 CAS (7 / 7) case and *n*proc is 64. *N* is the number of tasks in a given time interval.

linearly with *n*proc, as they should. The same behavior should be observed for the 2-external integrals. However, in this latter case, reading a block of integrals for a fixed set of internal orbital indices is avoided if there are no GUGA loops present for that particular **v** vector segment pair. Therefore, the actual amount of transferred data increases at a significantly reduced scale. The data transfer rates for reading the **v** vector and the 3- and 4-external integrals from GAs are about 30 MB/s. The rate for the **w** vector is smaller (~ 20 MB/s.) because the update process is more complex than the simple read. All data transfer speeds decrease somewhat with an increasing number of processors because of overlapping access requests coming from different processors to the same processor and because of possible data congestion effects in the network. The just-mentioned data rates refer to data stored as GAs. For data stored on local virtual disks (as, e.g., for the 0-, 1-, and 2-external integrals), much higher data access speeds of 100 MB/s and more are obtained be-

TABLE III.
Analysis of CRAY T3D Wall clock Timings for One Typical Davidson Iteration.

*n*proc	**Hv** Tot. (s)[a]	Data Transfer (s)			Data Transferred (MB)			Data Transfer Speed (MB / s)				Time for One Iter. (s.)	*e* (%)[b]
		v + **w** file	0-, 1-, 2-ext. integr.	3-, 4-ext. integr.	**v** + **w** file	0-, 1-, 2-ext. integr.	3-, 4-ext. integr.	**v**	**w**	0-, 1-, 2-ext. integr.	3-, 4-ext. integr.		
CH_3 CAS (7 / 7)[c]													
4	264	2.9	0.1	0.1	86	9	14	39	24	116	161	68.1	99.6
8	265	4.3	0.1	0.1	113	15	14	35	21	116	158	34.4	99.7
16	265	6.0	0.2	0.2	145	23	29	32	19	114	159	17.3	99.7
32	267	8.0	0.3	0.4	182	36	57	30	18	114	158	8.9	99.4
64	271	11.4	0.5	0.7	235	54	115	29	16	114	156	4.6	99.6
Butadiene CAS (4 / 5)[d]													
16	2275	42	0.6	1.4	965	65	47	32	18	112	33	150.2	99.0
32	2288	53	0.8	2.9	1241	86	93	31	19	110	32	76.2	98.8
64	2315	74	1.2	6.1	1599	130	186	30	17	106	30	38.8	99.3
128	2339	98	1.8	12.5	2069	176	373	29	16	101	30	20.0	99.3
Butadiene SR[e]													
1	20.0	0.05	0.04	0.15	7	5	24	170	110	109	161	20.7	100.0
2	20.4	0.24	0.08	0.15	9	9	24	51	30	118	160	10.7	100.0
4	21.0	0.40	0.12	0.15	12	14	24	41	23	119	160	5.6	99.5
8	22.0	0.66	0.17	0.30	16	20	49	34	19	119	161	3.0	97.4
16	23.7	1.01	0.24	0.54	21	28	86	31	16	117	160	1.6	98.9

[a] Data transfer time for **v** and **w** files not included.
[b] See eq. (5).
[c] All integrals on local virtual disks, the **v** and **w** vectors, and the diagonal elements of **H** as global arrays. The standard *n*tppr values were used.
[d] The 0-, 1-, and 2-external integrals on local virtual disks; the remaining two-electron integrals, the **v** and **w** vectors, and the diagonal elements of **H** as global arrays. The standard *n*tppr values were used.
[e] All integral files on local virtual disks; the **v** and **w** vectors, and the diagonal elements of **H** as global arrays; *n*tppr34 changed to 7 / 16, standard *n*tppr values otherwise.

cause of access to central memory. The efficiency of parallelization is very high (around 98–99% in most cases).

For the SR butadiene calculation the situation is different. The overall execution time is smaller than for the butadiene MR case by a factor of about 100. Therefore, one has to expect that SR butadiene will run efficiently only on a much smaller number of processors because there will not be sufficient work available that could be distributed to the processors. With our standard choice for the *n*tppr parameters the SR case works well only up to four processors. Optimization of load balancing with respect to *n*tplpr34 gave a value of 7/16 and led to a significant improvement. The results are shown in Table III. We obtained good scaling up to 16 processors. When proceeding to larger processor numbers the main problem occurs with the treatment of the 3- and 4-external integral cases. In our present scheme the respective integral file has to be read completely for each task dealing with the 3- and 4-external integral contributions. With 16 processors only a small number (about 16–20) of internal walks contribute to each **v** vector segment. This number will be reduced further with larger processor numbers. Thus, the ratio between the numerical operations to be carried out for each segment and the time for reading the integrals becomes less favorable. Even storing the 3- and 4-external integrals in local central memory does not help too much. A possible solution for this problem would be the replacement of the segment driven load balancing scheme for the 3- and 4-external integral cases by an integral driven one. This change is planned in conjunction with the implementation of a ''double direct'' parallel algorithm based on our AO-integral driven MRCI program[50–52] (T. Kovar, V. Parasuk, and H. Lischka, unpubl. work).

In Table IV a survey of the entire CI calculation section is given for butadiene CAS (4/5). For the purpose of a smooth comparison of total timings over all processor numbers, a subspace dimension of two was chosen because of memory restrictions in the global arrays for smaller processor numbers. With this subspace dimension the number of Davidson iterations is 26, using an energy convergence criterion of 10^{-6}, whereas only 12 iterations are required if the subspace is not restricted. Because the subspace operations are completely independent of the $\mathbf{H} \cdot \mathbf{v}$ section the computer time would be reduced in close proportion to the actual number of iterations needed in those cases where larger subspace dimensions are possible. The first iteration is not optimal because the task list is not ordered according to execution times. For the remaining iterations we obtain excellent parallel efficiencies. The time for the subspace operations is practically negligible. The speedups with respect to total times (based on the 16 processor calculation) are very satisfactory. A similar analysis carried out for the CH_3 case shows good speedups up to 64 processors.

In a single-processor calculation, the CI section of a MR calculation takes by far the largest fraction (about 91% for CH_3 and 99% for butadiene in terms of CPU time) of the total computation time. For the butadiene SR calculation only about 42% is consumed by the CI part. These percentages are fairly typical and illustrate the usefulness of the parallel CI code, especially for MR calculations.

To get an impression of what can be gained by further optimizing the load balancing, we adjusted the *n*tppr values empirically for the largest number of processors used. The results are presented in Figure 3 in the form of speedup curves for the CH_3, CAS (7/7) and butadiene CAS (4/5) calculations. From this figure one can find an increase in scalability by a factor of two. For CH_3 the calculation scales well up to 128 and for butadiene up to 256 processors. The speedup curve for a typical Davidson iteration is located closer to the theoretical curve than the one for the total time because of the startup time.

Investigations similar to the ones described for the CRAY T3D were also performed for the IBM SP2 and the Intel Paragon. In Table IV timings for the total CI calculation and in Table V timings for a single Davidson iteration are presented. These data show a very satisfactory performance of the COLUMBUS CI program also on these machines: one Davidson iteration for the butadiene CAS (4/5) case takes only about 20–30 s. We do not want to go into a detailed analysis of all data that, even though of great interest, would be too lengthy. We leave it to the interested reader to look into the results in more detail.

So far we have discussed applications where the data were stored either as GAs or on local virtual disks. In our last example we investigate the performance behavior when files are stored on disk for which the data transfer rate ($\sim$ 2 MB/s) is significantly reduced in comparison to the above-

TABLE IV.
Wall Clock Timings for Total CI Section for Butadiene CAS (4 / 5) Calculation.

*n*proc	Startup Time	First Iter.	Typ. Iter.	Subspace Oper.[b]	26 Iter.[c]	Total Time	Speedup Typ. Iter.	Speedup Tot. Time
				CRAY T3D[d]				
16	9.3	174.4	150.2	3.4	3916	3925	16	16
32	9.5	96.6	76.2	1.7	1994	2004	31.5	31.3
64	10.9	45.9	38.8	0.9	1014	1025	61.9	61.3
128	13.3	23.1	20.0	0.5	521	535	120.4	117.5
				IBM SP2[e]				
1	19.2	717.7	790.9	10.6	20,439	20,459	1	1
2	16.8	361.2	400.8	5.3	10,372	10,389	1.97	1.97
4	13.6	198.9	205.8	2.7	5339	5352	3.84	3.82
8	14.0	117.4	104.4	1.4	2715	2729	7.58	7.50
16	10.9	62.3	54.3	0.83	1414	1425	14.6	14.4
32	24.0	34.2	28.3	0.55	738	762	28.0	26.9
				Intel Paragon[d]				
32	48.4	156.8	114.5	2.3	3030	3079	32	32
64	44.5	67.3	59.4	1.2	1552	1597	61.7	61.7
96	47.2	48.9	41.4	0.8	1084	1131	88.5	87.1
128	48.8	39.8	32.9	0.6	862	911	111.3	108.1

Timings in seconds. The standard *n*tppr values were used for the calculation.
[a]Including subspace operations.
[b]Per iteration.
[c]10^{-6} accuracy in total energy according to ref. 56. A Davidson subspace of dimension 2 was used.
[d]0-, 1-, and 2-external integrals on local virtual disks; the remaining two-electron integrals, the **v** and **w** vectors, and the diagonal elements of **H** as global arrays.
[e]All integrals on local virtual disks; the **v** and **w** vectors and the diagonal elements of **H** as global arrays.

mentioned other storage devices. In Table VI timings for a calculation on butadiene CAS (4/5), using an extended basis set, are presented. The file size for the two-electron integrals is about 500 MB. The subspace dimensions of the Davidson scheme as given in Table VI were chosen according to the global memory available. In the first step calculations were carried out where the 3- and 4-external integrals were stored on disk. With this arrangement, the 32 and 64 processor calculations show a very good performance. Using 128 processors (results not given in Table VI) disk I/O concerning the 4-external integrals blocks the calculation. However, in this case there is already enough memory available to store these integrals as GAs. Therefore, we continued the series of calculations by shifting integrals in steps from disk to GAs. Inspection of Table VI shows very good scaling of the calculation up to 256 processors.

Conclusions and Outlook

We presented a massively parallel implementation of the CI part of the COLUMBUS program system. Calculations that take many hours on powerful workstations can be done within several minutes now. Besides obvious advantages for calculations on single geometries, systematic scans of energy surfaces will be greatly facilitated. Detailed timings were given to demonstrate the efficiency and scalability of our program. Recently, we carried out MRCI calculations on the first ^1B_u state of butadiene in connection with investigations on the $^1A_1/^1B_u$ excitation energy using a triple zeta basis set, including polarization and diffuse p functions with 38 million configurations. Still much larger calculations with up to 100 million CSFs seem

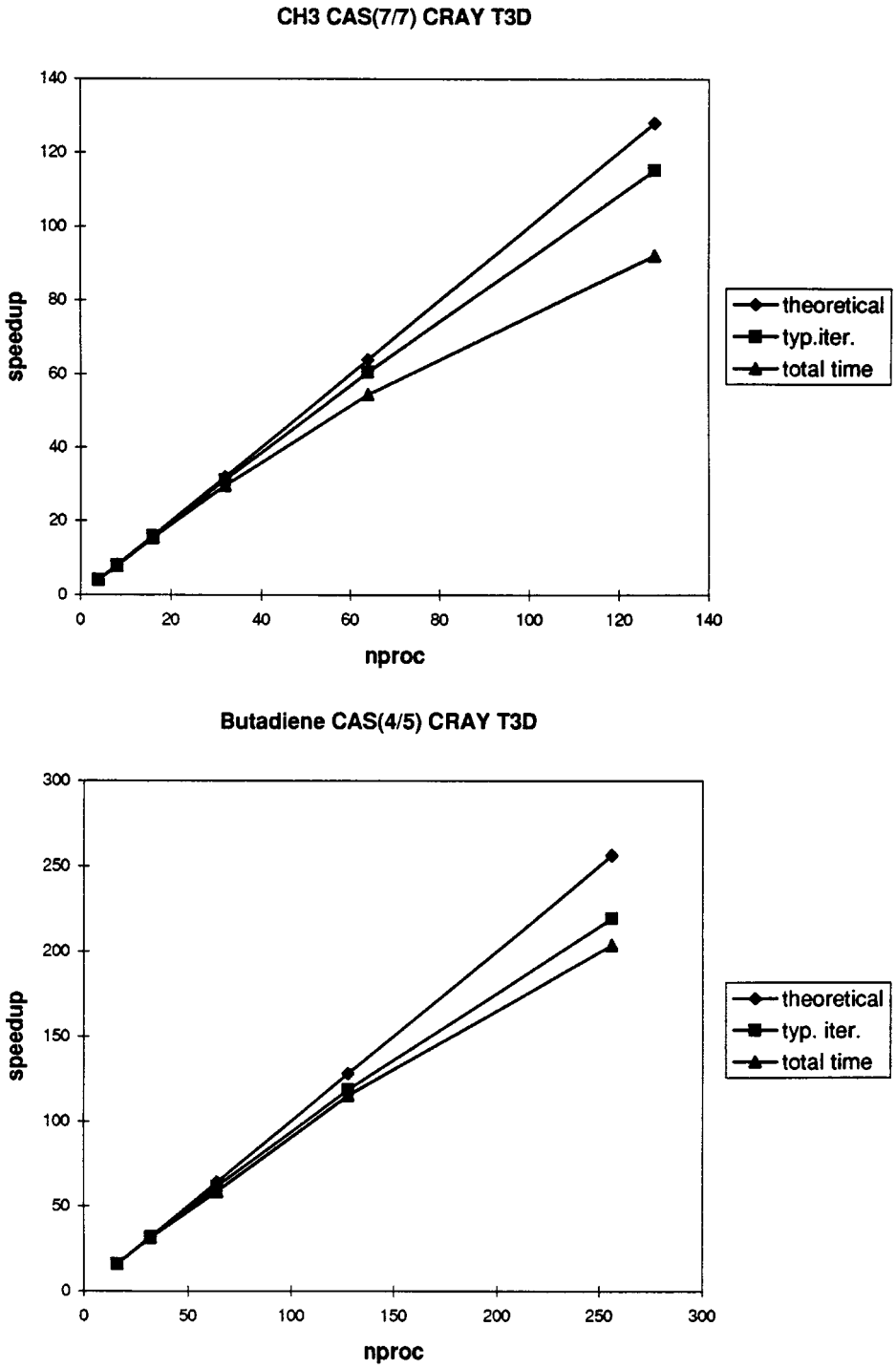

FIGURE 3. Speedup curves for the CRAY T3D obtained with optimized *n*tppr parameters. Wall clock times: CH_3 (CAS 4 / 5): four processors, typical iteration 68.8 s, total calculation 823.1 s; 128 processors, typical iteration 2.4 s, total calculation 35.7 s; butadiene CAS (4 / 5): 16 processors, typical iteration 149.5 s, total calculation 3904.1 s; 256 processors, typical iteration 10.9 s, total calculation 306.5 s.

TABLE V.
Analysis of Wall Clock Timings of One Typical Davidson Iteration for Butadiene CAS (4 / 5) Calculation.

*n*proc	**Hv** Tot. (s)	Data Transfer (s)			Data Transferred (MB)			Data Transfer Speed (MB / s.)				Time for One Iter. (s)	*e* (%)[a]
		v + **w** file	0-, 1-, 2-ext. integr.	3-, 4-ext. integr.	**v** + **w** file	0-, 1-, 2-ext. integr.	3-, 4-integr.	**v**	**w**	0-, 1-, 2-ext. integr.	3-, 4-ext. integr.		
						IBM SP2[b]							
1	776	2	0.1	0.2	308	7	23	183	178	102	106	790.9	100
2	777	8	0.1	0.2	370	11	23	43	51	112	107	400.8	99.5
4	782	14	0.2	0.2	430	18	23	28	34	116	105	205.8	98.5
8	786	20	0.2	0.2	551	26	23	25	30	116	105	104.4	98.5
16	790	29	0.3	0.4	672	37	47	22	25	114	106	54.3	96.8
32	799	39	0.5	0.9	853	54	93	20	25	106	107	28.3	96.9
						Intel Paragon[c]							
32	3476	47	2.3	12.4	853	54	93	14	26	24	7	114.5	98.7
64	3535	87	3.4	39.5	1130	83	186	9	26	24	5	59.4	98.2
96	3610	133	4.4	87.0	1342	108	279	7	21	24	3	41.4	97.6
128	3702	178	5.1	165.5	1500	123	373	5	21	24	2	32.9	95.9

The standard *n*tppr values were used for the calculation.
[a]See eq. (5).
[b]All integrals on local virtual disks; the **v** and **w** vectors and the diagonal elements of **H** as global arrays.
[c]The 0-, 1-, and 2-external integrals on local virtual disks; the remaining two-electron integrals, the **v** and **w** vectors, and the diagonal elements of **H** as global arrays.

TABLE VI.
Timings for Butadiene CAS (4 / 5) Calculation Using an Extended Basis Set.

*n*proc	Typ. Iter.	Total Time
32[a,b]	751	18,030 (2, 24)[c]
64[a,b]	381	9,214 (2, 24)[c]
128[b,d]	194	2,561 (4, 13)[c]
256[b,e]	92	1,210 (8, 12)[c]

Wall clock timings (s) performed on a CRAY T3D. Basis set: cc-*p*VTZ[49] (excluding *d* functions on the hydrogen atoms, 174 basis functions). No. of CFSs: 19,375,033.
[a] 3- and 4-external integrals (~ 483 MB) on disk; remaining integrals as global arrays.
[b] **v** and **w** vectors and the diagonal elements of **H** as global arrays.
[c] Subspace dimension and number of Davidson iterations in parentheses.
[d] 3-external integrals on disk; remaining integrals as global arrays.
[e] 3- and 4-external integrals as global arrays; remaining integrals on virtual disk.

possible. This feature is certainly a valuable extension of the capabilities of full CI calculations[53–55] that are restricted to rather small molecules and basis sets.

We also discussed the restrictions that still apply to our present version. The main one is that the 3- and 4-external two-electron integrals should be kept preferentially as GAs distributed over the memory of the processors. However, as our last example showed, storing these integrals on disk also gives good performance up to a reasonably large number of processors and thus removes some of the restrictions of the program with respect to larger basis sets. However, to get more flexibility for the program, we are going to change the algorithm by introducing an AO-driven scheme[50–52] for the handling of the 3- and 4-external integrals and recalculating the AO integrals every time (double direct CI). Up to now they have had to be transformed completely to the MO basis first. An AO-driven, sequential program is already available (T. Kovar and H. Lischka, unpubl. work). Bypassing the full integral transformation is also one key step for the parallelization of the whole MRCI energy calculation. In connection with the 100 million CSF goal we should also mention our plans to introduce into the parallel program a data compression scheme for the **v** and **w** vectors that was developed by us recently.[56] Data compression will be necessary to store large expansion vectors as GAs. Work is also in progress to add a swap file to a GA to give more flexibility to the memory management.

The portability of the program was demonstrated by applications on three major parallel computer systems. A port to the Silicon Graphics Power Challenge is underway. Last, but not least, we should mention that the program also works on clusters of Unix workstations. In fact, this is our major platform for developing and debugging the program code. However, at least in an ethernet environment, the communication rate between workstations is too low. Moreover, our load balancing scheme is most efficient if exclusive access to the individual nodes is available, a situation that is rarely encountered in a workstation environment. Therefore, we did not pursue further the possibilities for using workstation clusters for actual production work.

The parallel COLUMBUS program (as well as the sequential program system) is available for general use. Send any requests to lischka@itc.univie.ac.at.

Author's Note

The timings obtained with the parallel COLUMBUS program system presented in this article represent preliminary work. The individual codes have been optimized to different extents on the various computer systems. Consequently, these timings cannot be used directly to assess either the ultimate performance of the COLUMBUS program system or to compare the performance of the different computer systems.

Acknowledgments

This work was performed under the auspices of the Austrian Fonds zur Förderung der Wissenschaftlichen Forschung, Projects P9032-CHE and P10681-CHE; the High Performance Computing and Communication Program of the Office of Scientific Computing, U.S. Department of Energy, under contract DE-AC06-76RLO 1830 with Battelle Memorial Laboratory, which operates the Pacific Northwest National Laboratory; and the Office of Basic Energy Sciences, Division of Chemical Sciences, U.S. Department of Energy, under contract W-31-109-ENG-38 with the University of Chicago, which operates the Argonne National Laboratory. This research was performed in part using the

Intel Paragon operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Pacific Northwest National Laboratory. Access to the CRAY T3D was provided by the Massively Parallel Processing Access Program of the National Energy Research Supercomputer Center and the Division of Mathematical, Information, and Computational Sciences, Office of Computational and Technology Research, U.S. Department of Energy. This research was sponsored in part by the Phillips Laboratory, Air Force Material Command, USAF, through the use of the MHPCC under Cooperative Agreement F29601-93-2-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Phillips Laboratory or the U.S. Government.

References

1. The MPI Forum, In *Proceedings of Supercomputing '93*, IEEE Computer Society Press, Los Alamitos, CA, 1993, p. 878.
2. High Performance Fortran Forum, *Technical Report Version* 1.0, William Marsh, Rice University, Houston, TX, 1993.
3. E. Clementi, G. Corongiu, J. Detrich, S. Chin, and L. Domingo, *Int. J. Quantum Chem. Symp.* **18,** 601 (1984).
4. J. E. Watts, M. Dupuis, and H. O. Villar, Technical Report, KGN-78, IBM, Kingston, NY, 1986.
5. M. Dupuis and J. D. Watts, *Theor. Chim. Acta*, **71,** 91 (1987).
6. G. Corongiu and J. H. Detrich, *IBM J. Res. Dev.*, **29,** 422 (1985).
7. R. J. Harrison and R. A. Kendall, *Theor. Chim. Acta*, **79,** 337 (1991).
8. H. P. Lüthi, J. E. Mertz, M. W. Feyereisen, and J. E. Almlöf, *J. Comput. Chem.*, **13,** 160 (1992).
9. S. Kindermann, E. Michel, and P. Otto, *J. Comput. Chem.*, **13,** 414 (1992).
10. M. W. Feyereisen, R. A. Kendall, J. Nichols, D. Dame, and J. T. Golab, *J. Comput. Chem.*, **14,** 818 (1993).
11. S. Brode, H. Horn, M. Ehrig, D. Moldrup, J. E. Rice, and R. Ahlrichs, *J. Comput. Chem.*, **14,** 1142 (1993).
12. M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery, Jr., *J. Comput. Chem.*, **14,** 1347 (1993).
13. M. E. Colvin, C. L. Janssen, R. A. Whiteside, and C. H. Tong, *Theor. Chim. Acta*, **84,** 301 (1993).
14. L. G. M. Petterson and T. Faxen, *Theor. Chim. Acta*, **85,** 345 (1993).
15. A. Burkhardt, U. Wedig, and H. G. v. Schnering, *Theor. Chim. Acta*, **86,** 497 (1993).
16. T. R. Furlani and H. F. King, *J. Comput. Chem.*, **16,** 91 (1995).
17. I. T. Foster, J. L. Tilson, A. F. Wagner, R. L. Shepard, R. J. Harrison, R. A. Kendal, and R. J. Littlefield, *J. Comput. Chem.*, **17,** 109 (1996).
18. R. J. Harrison, M. F. Guest, R. A. Kendall, D. E. Bernholdt, A. Wong, M. Stave, J. L. Anchell, A. C. Hess, R. J. Littlefield, G. L. Fann, J. Nieplocha, G. S. Thomas, D. Elwood, J. L. Tilson, R. L. Shepard, A. F. Wagner, I. T. Foster, E. Lusk, and R. Stevens, *J. Comput. Chem.*, **17,** 124 (1996).
19. J. D. Watts and M. Dupuis, *J. Comput. Chem.*, **9,** 158 (1988).
20. A. M. Márquez and M. Dupuis, *J. Comput. Chem.*, **16,** 395 (1995).
21. I. M. B. Nielsen and E. T. Seidl, *J. Comput. Chem.*, **16,** 1301 (1995).
22. A. P. Rendell, T. J. Lee, and R. Lindh, *Chem. Phys. Lett.*, **194,** 84 (1992).
23. A. P. Rendell, M. F. Guest, and R. A. Kendall, *J. Comput. Chem.*, **14,** 1429 (1993).
24. R. A. Whiteside, J. S. Binkley, M. E. Colvin, and H. F. Schaefer, III, *J. Chem. Phys.*, **86**, 2185 (1987).
25. T. L. Windus, M. W. Schmidt, and M. S. Gordon, *Theor. Chim. Acta*, **89,** 77 (1994).
26. R. J. Harrison and E. A. Stahlberg, In *High Performance Computing and Its Applications in the Physical Sciences*, D. A. Browne, J. Callaway, J. P. Draayer, R. W. Haymaker, R. K. Kalia, J. E. Tohline, and P. Vashishta, Eds., World Scientific, Singapore, 1993, p. 176.
27. G. L. Bendazzoli and S. Evangelisti, *J. Chem. Phys.*, **98,** 3141 (1993).
28. S. Evangelisti, G. L. Bendazzoli, R. Ansaloni, and E. Rossi, *Chem. Phys. Lett.*, **233,** 353 (1995).
29. T. J. Martinez and E. Carter, *Chem. Phys. Lett.*, **241,** 490 (1995).
30. R. J. Harrison and R. Shepard, *Annu. Rev. Phys. Chem.*, **45,** 623 (1994).
31. R. A. Kendall, R. J. Harrison, R. J. Littlefield, and M. F. Guest, In *Reviews in Computational Chemistry VI*, K. B. Lipkowitz and D. B. Boyd, Eds., VCH Publishers, New York, 1995.
32. M. Schüler, T. Kovar, H. Lischka, R. Shepard, and R. J. Harrison, *Theor. Chim. Acta*, **84,** 489 (1993).
33. H. Lischka, H. Dachsel, R. Shepard, and R. J. Harrison, In *High Performance Computing and Networking*, Lecture Notes in Computer Science *796*, W. Gentzsch and U. Harms, Eds., Springer, New York, 1994, p. 203.
34. H. Lischka, H. Dachsel, R. Shepard, and R. J. Harrison, In *Parallel Computing in Computational Chemistry*, ASC Symposium Series *592*, T. G. Mattson, Eds., ACS, Washington, D.C., 1995.
35. H. Lischka, R. Shepard, F. Brown, and I. Shavitt, *Int. J. Quantum Chem.*, **S15,** 91 (1981).
36. R. Ahlrichs, H.-J. Böhm, C. Erhard, P. Scharf, H. Schiffer, H. Lischka, and M. Schindler, *J. Comput. Chem.*, **6,** 200 (1985).
37. R. Shepard, I. Shavitt, R. M. Pitzer, D. C. Comeau, M. Pepper, H. Lischka, P. G. Szalay, R. Ahlrichs, F. B. Brown, and J. G. Zhao, *Int. J. Quantum Chem.*, **S22,** 149 (1988).
38. R. J. Gdanitz and R. Ahlrichs, *Chem. Phys. Lett.*, **143,** 413 (1988).
39. P. G. Szalay and R. J. Bartlett, *J. Chem. Phys.*, **103,** 3600 (1995).
40. J. Paldus, In The Unitary Group for the Evaluation of Electronic Energy Matrix Elements, J. Hinze, Ed., Springer–Verlag, Berlin, 1981, p. 1.

41. I. Shavitt, In The Unitary Group for the Evaluation of Electronic Energy Matrix Elements, *J. Hinze*, Ed., Springer–Verlag, Berlin, 1981, p. 51.
42. E. R. Davidson, *J. Comp. Phys.*, **17,** 87 (1975).
43. B. O. Roos, *Chem. Phys. Lett.*, **15,** 153 (1972).
44. B. O. Roos and P. E. M. Siegbahn, In *Methods of Electronic Structure Theory*, H. F. Schaefer, III, Ed., Plenum, New York, 1977, p. 277.
45. P. E. M. Siegbahn, In The Unitary Group for the Evaluation of Electronic Energy Matrix Elements, *J. Hinze*, Ed., Springer–Verlag, Berlin, 1981, p. 119.
46. I. Shavitt, Annual Report on New Methods in Computational Quantum Chemistry and Their Appication on Modern Super-Computers, Battelle Columbus Laboratories, Columbus, OH, 1979.
47. R. J. Harrison, *Int. J. Quantum Chem.*, **40,** 847 (1991).
48. (a) J. Nieplocha, R. J. Harrison, and R. J. Littlefield, In *Proceedings of Supercomputing 1994*, IEEE Computer Society, Washington, D.C., 1994, p. 340; (b) J. Nieplocha, R. J. Harrison, and R. J. Littlefield, *J. Supercomput.*, in press.
49. T. H. Dunning, Jr., *J. Chem. Phys.*, **90,** 1007 (1989).
50. R. Ahlrichs, In *Proceedings of the 5th European Seminar on Quantum Chemistry*, T. H. van Duijnen and W. C. Nieuwpoort, Eds., Max Planck Institute, Garching, Germany, 1982.
51. H.-J. Werner and E. A. Reinsch, *J. Chem. Phys.*, **76,** 3144 (1982).
52. P. Taylor, Int. *J. Quantum Chem.*, **XXXI,** 521 (1987).
53. P. Saxe, H. F. Schaefer, III, and N. C. Handy, *Chem. Phys. Lett.*, **79,** 202 (1981).
54. P. J. Knowles, *Chem. Phys. Lett.*, **155,** 513 (1989).
55. J. Olson, P. Jørgensen, and J. Simons, *Chem. Phys. Lett.*, **169,** 463 (1990).
56. H. Dachsel and H. Lischka, *Theor. Chim. Acta*, **92,** 339 (1995).